ETL Pipeline for RevStore Database Project

**(Data Extract, Transformation and Loading)**

|  |  |  |
| --- | --- | --- |
|  | **Contents** | **PgNo.** |
| **1.** | **Introduction**   * 1. **Problem Statement**   2. **Objective and Goals**   3. **Scope of Project** |  |
| **2.** | **Architecture** |  |
| **3.** | **Schema Design** |  |
| **4.** | **Methodology**   1. **Data Extraction**    1. **Identification and Collection**    2. **Reading the JSON Files** 2. **EDA** 3. **Data Transformation**    1. **Schema Mapping**    2. **Handling Missing and Inconsistent Data**    3. **Normalization Process** 4. **Data Loading**    1. **Database Connectivity**    2. **Insertion into Tables:** 5. **Logging and Error Handling** |  |
| **5.** | **Challenges and Solutions** |  |
| **6.** | **Summary** |  |

**1. Introduction**

**1.1 Problem Statement**

You have been provided with a collection of JSON files containing complex and nested data structures related to customer transactions. The data needs to be transformed into a normalized, tabular format to enable seamless integration with an existing MySQL database. The challenge lies in handling the nested and hierarchical nature of the JSON data and ensuring that the resulting tables adhere to the database schema.

**1.2 Objectives**

The primary objective of this project is to develop a robust ETL (Extract, Transform, Load) pipeline that processes and normalizes JSON data files, transforming them into a structured format suitable for integration into a MySQL database. The specific goals are as follows:

* Data Extraction: Load and aggregate JSON data from multiple files.
* Data Normalization: Flatten and normalize nested JSON structures.
* Data Transformation: Apply business logic and ensure data integrity.
* Data Loading: Insert the normalized data into MySQL tables.
* Data Testing: Test the data after loaded into Db.

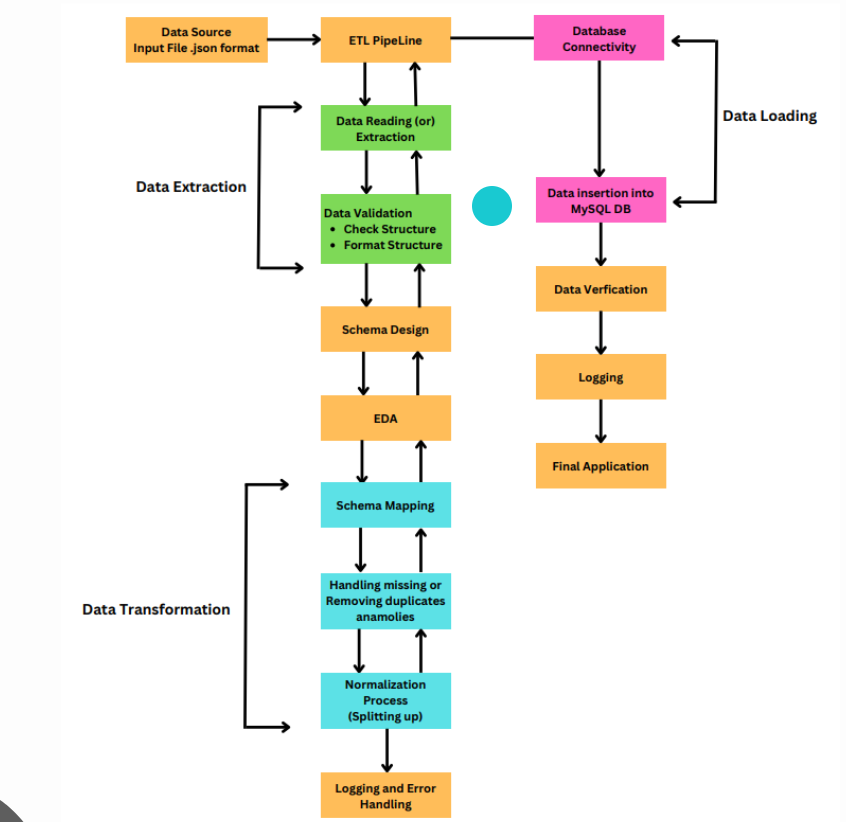
**1.3 Scope of the Project**

This project encompasses the entire process of JSON data normalization and ETL pipeline implementation, from data extraction to database integration. The scope includes:

* Reading and processing JSON files containing customer transaction data.
* Normalizing nested JSON structures into a flat table format.
* Creating separate DataFrames/tables for different entities (Customers, Orders, Products, etc.).
* Transforming and cleaning the data as per the business requirements.
* Loading the cleaned and normalized data into a MySQL database.

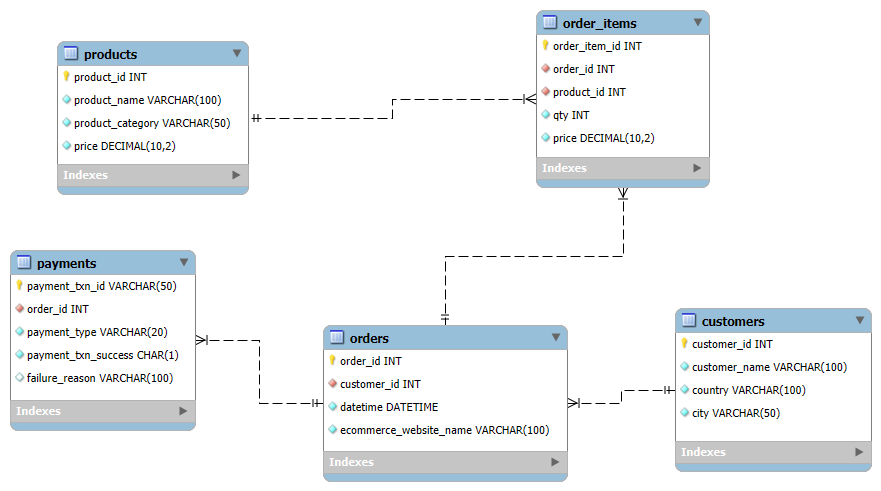
**Out of scope are data analysis, visualization, and any front-end interface development.**

**2. Architecture:**

****

**3. Schema Design**

**ER Diagrams:**

****

**3.1 Customers Table**

* **Fields:**
  + customer\_id (Primary Key): A unique identifier for each customer, usually a numeric or alphanumeric code.
  + customer\_name: The full name of the customer.
  + city: The city where the customer resides.
  + state: The state where the customer resides.
  + country: The country where the customer resides.
* **Description:**  
  The Customers table stores detailed information about each unique customer, including their location details. This table serves as the foundation for customer-related data and is linked to other tables (e.g., Orders, Payments) via the customer\_id field. By capturing geographic information like city, state, and country, this table supports location-based analysis and reporting.
* **Validations**:
  + Uniqueness: The customer\_id must be unique across the table to prevent duplicate customer entries.
  + Non-null constraints: Fields like customer\_id, customer\_name, and country should be non-null, as they are essential for identifying and categorizing customers.
  + Data Type: The customer\_id should be of a consistent data type (e.g., integer or string), while city, state, and country should be strings.
  + Length Constraint: customer\_name may have a length constraint to ensure data consistency, e.g., a maximum of 100 characters.

**3.2 Orders Table**

* **Fields:**
  + order\_id (Primary Key): A unique identifier for each order, typically a numeric or alphanumeric code.
  + customer\_id (Foreign Key): References the customer\_id in the Customers table, linking the order to the specific customer.
  + datetime: The date and time when the order was placed.
  + ecommerce\_website\_name: The name of the e-commerce website where the order was placed.
* **Description:**  
  The Orders table stores critical information about each order placed by customers. This includes the time and platform of the transaction, which is essential for tracking purchase behaviors and trends. The table’s relationship with the Customers table allows for detailed customer order history and insights.
* **Validations:**
  + Uniqueness: The order\_id must be unique across the table to ensure that each order is recorded only once.
  + Non-null constraints: Fields like order\_id, customer\_id, and datetime should not be null, as they are essential for tracking orders.
  + Foreign Key Constraint: The customer\_id should be a valid reference to an existing customer\_id in the Customers table.
  + Date Format: The datetime field should follow a standard date-time format (e.g., YYYY-MM-DD HH:MM:SS) to allow for accurate sorting and filtering.

**3.3 Products Table**

* **Fields:**
  + product\_id (Primary Key): A unique identifier for each product.
  + product\_name: The name of the product.
  + product\_category: The category or type of the product (e.g., Electronics, Apparel).
* **Description:**  
  The Products table contains information about all products available for purchase. This table is crucial for understanding what products are sold, their categories, and enabling analysis of sales patterns by product type.
* **Validations:**
  + Uniqueness: The product\_id must be unique to ensure each product is distinctly identifiable.
  + Non-null constraints: Fields like product\_id and product\_name should be non-null.
  + Data Type: product\_id should be consistently typed (e.g., integer or string), and product\_category should be a string.
  + Length Constraint: product\_name and product\_category may have length constraints (e.g., maximum 100 characters) to maintain data integrity.

**3.4 Order\_Items Table**

* **Fields:**
  + order\_id (Foreign Key): References the order\_id in the Orders table, linking each item to a specific order.
  + product\_id (Foreign Key): References the product\_id in the Products table, linking the item to a specific product.
  + qty: The quantity of the product ordered.
  + price: The price of the product at the time of the order.
* **Description:**The Order\_Items table represents the detailed breakdown of items within each order. It connects orders to products and includes the quantity and price of each item, making it a critical component for financial and sales analysis. This table enables detailed insights into what products are selling, at what prices, and in what quantities.
* **Validations:**
  + Composite Key: The combination of order\_id and product\_id can serve as a composite primary key to ensure that the same product isn’t listed multiple times in a single order.
  + Foreign Key Constraints: Both order\_id and product\_id should reference valid entries in their respective tables.
  + Non-null constraints: Fields like order\_id, product\_id, qty, and price should not be null.
  + Positive Quantity: The qty field must be a positive integer, as negative quantities would not make sense in this context.
  + Price Validation: The price field should be a positive number, and may require precision (e.g., two decimal places for currency).

**3.5 Payments Table**

* **Fields:**
  + payment\_txn\_id (Primary Key): A unique identifier for each payment transaction.
  + order\_id (Foreign Key): References the order\_id in the Orders table, linking the payment to a specific order.
  + payment\_type: The method of payment used (e.g., Credit Card, PayPal).
  + payment\_txn\_success: A boolean or status indicator showing whether the transaction was successful or not.
* **Description:**  
  The Payments table stores details about payment transactions, including their success status and the payment method used. This table is essential for tracking the financial aspect of orders, verifying successful transactions, and analyzing payment method preferences.
* **Validations:**
  + Uniqueness: The payment\_txn\_id must be unique to ensure that each payment transaction is distinctly identifiable.
  + Foreign Key Constraint: The order\_id should reference a valid order\_id in the Orders table.
  + Non-null constraints: Fields like payment\_txn\_id, order\_id, and payment\_type should not be null.
  + Boolean Constraint: The payment\_txn\_success field should be a boolean or enumerated type to indicate the transaction status clearly (e.g., True/False or Success/Failure).

**4. Methodology**

The methodology of this project focuses on the systematic processing of JSON data into a normalized relational database schema using Python. The approach follows the standard Extract, Transform, Load (ETL) framework, which is methodically adapted to suit the specific requirements of the project. Below is a detailed breakdown of the methodology:

1. **Data Extraction**

* 1.1 Identification and Collection:
  + The first step involves identifying the relevant JSON files that contain the customer and transaction data. These files are stored within a designated directory structure.
  + A recursive function is developed to traverse through the directory, collecting the paths of all JSON files. This function is crucial for ensuring that no relevant file is missed, even if stored in nested subdirectories.
* 1.2 Reading the JSON Files:
  + Once the paths are collected, the JSON files are read into Python. The JSON data is loaded into appropriate data structures, such as dictionaries or Pandas DataFrames, depending on the data's nature and the intended processing steps.

**2. Exploratory Data Analysis (EDA):**

* Goal: Gain insights into the structure, distribution, and relationships within the data.
* Steps:
  + Understanding the Data Structure: Look at the data types, columns, and the first few rows to understand the basic structure.
  + Summary Statistics: Generate summary statistics (mean, median, mode, standard deviation, etc.) to get an overview of each column.
  + Data Visualization: Create visualizations like histograms, bar charts, box plots, and scatter plots to visualize distributions, outliers, and correlations.
  + Handling Missing Values: Identify missing values and decide on strategies for handling them (e.g., removal, imputation).
  + Detecting Outliers: Identify and analyze outliers to understand if they represent errors or significant variations.
  + Correlation Analysis: Check for correlations between different columns to understand relationships.
  + Categorical Data Analysis: For categorical data, examine the frequency distribution and relationships with other variables.
* Outcome: The EDA process helps you decide on necessary transformations, identify any data quality issues, and prepare the data for loading into the database.

**3. Data Transformation**

* 3.1 Schema Mapping:
  + The raw JSON data is mapped to the predefined schema tables: Customers, Orders, Products, Order\_Items, and Payments.
  + Data transformation functions are created to extract specific fields from the JSON files and map them to the corresponding fields in the relational schema.
  + Data types are validated and converted as necessary, ensuring that each field adheres to the schema's constraints (e.g., converting dates to a standard format, ensuring numeric fields are correctly typed).
* 3.2 Handling Missing and Inconsistent Data:
  + During transformation, checks are performed to identify and handle any missing or inconsistent data. Missing values are either filled with default values, left as null, or the record is flagged for further review, depending on the business logic.
  + Inconsistent data, such as mismatches in field types or invalid references (e.g., an order\_id that does not exist in the Orders table), are logged for analysis and corrective actions.
* 3.3 Normalization Process:
  + The data is normalized according to the principles of database normalization, primarily focusing on the third normal form (3NF). This involves breaking down the data into the appropriate tables (e.g., separating product details from order details) to eliminate redundancy and ensure data integrity.
  + Foreign keys are assigned to establish relationships between tables, such as linking Orders with Customers and Order\_Items with Products.

**4. Data Loading**

* 4.1 Database Connectivity:
  + The normalized data is loaded into a MySQL database. A connection to the database is established using Python's database connectors (e.g., MySQL Connector/Python or SQLAlchemy).
  + The connection parameters, such as host, username, password, and database name, are securely managed to ensure a safe and reliable connection.
* 4.2 Insertion into Tables:
  + Insert operations are performed for each normalized table (Customers, Orders, Products, Order\_Items, Payments).
  + Bulk inserts are used where appropriate to improve performance, especially when dealing with large datasets.
  + The operations are wrapped in transactions to ensure atomicity, meaning that all inserts for a particular dataset are either fully completed or fully rolled back in case of an error.

**5. Logging and Error Handling**

* 5.1 Logging:
  + A robust logging mechanism is implemented to record all significant events during the ETL process. This includes data extraction start and end times, the number of records processed, any anomalies encountered, and confirmation of successful data loading.
  + Logs are categorized by severity (e.g., INFO, DEBUG, ERROR) to facilitate quick identification of critical issues.
* 5.2 Error Handling:
  + Comprehensive error handling is implemented to manage unexpected issues during extraction, transformation, or loading. Exceptions are caught and logged, with appropriate actions taken, such as retrying the operation or halting the process if necessary.
  + Errors that prevent successful data loading are communicated to the development or operations team for resolution.

**5. Challenges and Solutions**

**5.1 Challenge 1: Handling Nested JSON Structures**

Problem: JSON data often contains nested structures, making it difficult to extract and flatten the data into a relational format.

Solution: The pandas.json\_normalize() function is used to flatten the JSON data, but special care must be taken to handle deeply nested fields. Custom functions are sometimes needed to recursively normalize these structures.

**5.2 Challenge 2: Ensuring Data Integrity**

Problem: Maintaining referential integrity between the normalized tables is crucial, especially when dealing with foreign key relationships.

Solution: During the data transformation process, the integrity of the relationships is preserved by ensuring that all primary and foreign keys are correctly assigned before loading the data into the database.

**5.3 Challenge 3: Handling Incomplete or Inconsistent Data**

Problem: JSON files may contain missing or inconsistent data, which can lead to errors during normalization or database insertion.

Solution: A robust data cleaning process is implemented, which includes filling in missing values, discarding incomplete records, and validating data types before loading it into the database.

**5.4** **Challenges Faced: Handling Duplicates with Primary Keys**

When inserting data into a database, one common challenge is handling duplicate entries, especially when a primary key constraint is in place. A primary key uniquely identifies each record in a table, so any attempt to insert a duplicate primary key value will result in an error. This can disrupt the data loading process, especially in large datasets.

Solution 1: Checking for Duplicates Before Inserting: This method is useful when you want to have more control over how duplicates are handled, such as skipping the insertion or performing a custom action if a duplicate is found.

Solution 2: Using ON DUPLICATE KEY UPDATE: This method is simpler and more efficient when you want to ensure that the database always contains the latest information, automatically updating existing records if a duplicate is detected.

**6. Conclusion**

This project successfully addresses the challenges of normalizing complex JSON data structures and integrating them into a relational database. By following a systematic approach involving data extraction, normalization, transformation, and loading, the project delivers a robust ETL pipeline that not only meets the immediate needs but also provides a scalable solution for future data processing tasks.

The outcome is a well-structured database that allows for efficient querying and analysis, laying the foundation for insightful decision-making by the business analytics team.